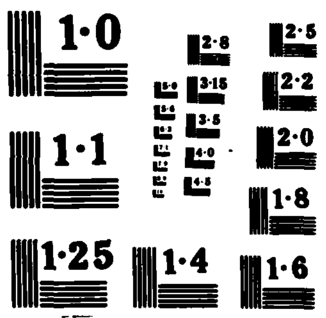

1·0
2·8
2·5
3·15
2·2
3·5
2·0
4·0
1·1
4·5
1·8
1·25
1·4
1·6

AD-A166 398

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Report No 85015

Title: PerqFlex Firmware

Authors: I F Currie, J M Foster and P W Edwards

Date: December 1985

Summary

This report describes the instruction set and general firmware architecture of the Flex computer as implemented on the ICL Perq workstation.

PerqFlex Firmware

Contents

Introduction

This report describes the firmware architecture of the Flex computing system as implemented on the ICL Perq. This implementation is very similar to that described in RSRE Report 81009, (Flex Firmware by Currie, Edwards and Foster) generally referred to as the Logica implementation. This is rather a misnomer, since although Logica built the hardware (to an RSRE design) on which it was implemented, Logica had nothing to do with the micro-coding or software construction of the Flex architecture.

The differences between the Perq and Logica implementation arise mainly from the properties of the underlying hardware. In the Logica Flex, for example, each byte in main-store was in fact nine bits long rather than eight; the extra bits (the tag bits) were used to distinguish between pointers and non-pointers ie between capabilities and scalar data. On Perq, however, there are no bits to spare like this. In order to distinguish between pointers and non-pointers in Perq,it was necessary to steal a bit from each 32-bit word to act as a tag bit. One effect of this is that integer arithmetic on PerqFlex is 31-bits long. Another more serious one from the point of view of compatibility with LogicaFlex is that multiple characters and bits can only reside in special blocks so that four characters packed into one word, for example, cannot be confused with a capability. This means that the uniform scheme, in the LogicaFlex, of being able to load any number of words,characters or bits into the register has had to be modified so that although one can still load any number of words into the register, only single characters or bits can be loaded in PerqFlex. This, in turn, means that extra instructions have been introduced into PerqFlex to allow one to manipulate characters and booleans in other ways.

Other differences arise from the treatment of peripheral devices on Perq. For example, Perq has a bit-mapped display with some specialised hardware for moving about rectangles of bits. Other external devices includes Winchester and floppy discs and GPIB, RS232 and Ethernet interfaces, all of which requires some specialised instructions.

In spite of these differences, program transfer between the Flex implementations is relatively painless. Most of the detailed differences in programs are coped with by the compilers. Other differences are minimised by the fact that most of the user's interfaces (eg to filestore) are the same and that Flex encourages a highly modular approach to program design.

1 Data in PerqFlex

Data in Flex can be handled in words, characters or booleans. Words are either pointers or non-pointers with distinct representations so that arithmetic cannot be performed on pointers and non-pointers cannot be used in indirect addressing operations. The pointers are used to represent capabilities to data or actions in main-store.

In blocks of main-store which can contain pointers, the least significant bit of each 32-bit word is used to distinguish pointers from non-pointers; a pointer has this tag bit set while a non-pointer has it clear.

1.1 Non-pointers

The various instructions which operate on non-pointers use the following representations:

Bool - 1 bit.

Char - 8 bits.
Packed Chars and Bools can only appear in blocks of type 3 and 11 respectively (see 1.2.1). When a single Char or Bool is pushed or stored in another type of block, it is stored as an Int.

Int - 1 32-bit word with unset ls bit.

31 bit 2's complement	tag bit=0

LongInt - 2 Ints, 1st Int is most significant and second is positive, giving 61 bits 2's complement.

word1:	0	30 bit unsigned	tag bit=0
word0:	31 bit 2's complement		tag bit=0

ShortReal - 1 word with unset ls bit. Biased 2's exponent in next 7. 24-bit mantissa with sign equal to the complement of the ms bit.

24 bit mantissa	7 bit biased exponent	tag bit=0

Real - 2 words with unset ls bits. The ms half of Word 0 is the ms 16 bits of the mantissa with sign equal to the complement of the ms bit. Next 4 bits are zero with biased 2's exponent in next 11. Next 31 bits of mantissa in ms end of Word 1.

Word 1:	31 bit mantissa extension (ls part)			tag bit=0
Word 0:	16 bit mantissa	0000	11 bit biassed exponent	tag bit=0

1.2 Pointers

A pointer is a 32-bit word with its least significant bit set. Its complete format is:

L	2 bit gc	0	4 bit type	S	3 bit spare	19 bit block address	tag bit=1

where the the block address is expressed in 32-bit word units and the type field gives the type of block pointed at. Each block has an overhead word which gives the size of the block and repeats its type. The gc bits both in pointers and overhead words are used for garbage collection purposes. Existing pointers can be copied freely but one cannot synthesize a pointer to an existing block, and the contents of a block can only be accessed according to rules defined by both the pointer and the type of the block. Even given maximum access, a pointer only allows access within the limits of its block, not including the overhead word. The access rules are modified if the pointer is locked (given by the L-bit = 1). The effect of locking for blocks of type other than 5 or 6 (see 1.2.1) is to make the blocks read-only. A pointer with its S bit = 1 is a shaky pointer, otherwise it is a firm pointer. This means that so long as a block is pointed at by a firm pointer then any shaky pointers remain valid. However, if there are no firm pointers to a block then any shaky pointers to it will be replaced by the non-pointer 0 in garbage collection and the block will vanish. This mechanism is largely intended to provide easy aliasing between disc and mainstore; it is also used internally in the procedure call mechanism.

1.2.1 Blocks

Each block starts on a 32-bit word boundary with one overhead word:

3 bit gc	1	4 bit type	4 bit spare	19 bit block size	tag bit=1

The gc bits are used for garbage collection purposes. The size is measured in 32-bit words, with the overhead word included.

There are in fact seven different types of block:

Type 1
This is a work-space block i.e. the locals of some call of a procedure, including pointers. The first four words of a work-space block are completely inaccessible to everybody but the micro-code. They contain link information for exiting from the procedure or for re-instating this work-space as the current locals. The instructions which access within a work-space block (e.g. opcodes 0 or 3) will automatically compensate for these extra invisible words so that, for example, a zero displacement (a-field) will in fact give the 5-th word of the block.

Type 2
This is a codeblock which points to the instructions (a type 3 block) and the constants (a type 4 block) of some procedure. It also contains a word giving the size of the workspace required to run this procedure. In addition, another word maintains a shaky chain of workspaces which may be used when the procedure is called in appropriate cases

Type 3
This block contains no pointers, each word in it being a full 32 bit quantity. Conventionally, it will contain packed 8-bit characters; this is supported by the dereference and packing instructions.

Type 4
This is a normal data block which can contain pointers.

Type 5
This is a closure block, containing pointers to a codeblock and a non-locals block. A pointer to a closure is a procedure. The only operations allowed on procedures are calls i.e. the contents of the closure are hidden. When a locked procedure is called, the resulting call will be run in privileged state.

Type 6
This is a keyed block whose access is controlled by knowledge of first word in block. Access to a keyed block via an unlocked pointer makes it identical to a normal data block; access via a locked pointer is barred. A locked pointer to a keyed block may be unlocked if one knows the contents of the first word of the block using instruction 160. Locked pointers to other types of blocks cannot be unlocked.

Type 11
This block contains no pointers. It is conventionally used to contain packed booleans. The bit-map for the screen is contained in such a block.

1.2.2 References, vectors and arrays

Since pointers only refer to complete blocks, several of the instructions make use of references i.e. (pointer, non-pointer) pairs with the interpretation that the first word gives a block and the second some kind of a displacement within it, subject to the usual access constraints of the pointer. The kind of displacement is defined by the type of block pointed at ie:

types 1,2,4,6 - word displacement.
type 3 - char displacement from logical start of block,
type 11 - bit displacement from logical start of block.

The contents of type 3 blocks cannot be examined and so cannot form part of a reference.

A vector is defined as a triple consisting of (non-pointer, reference) where the first word is the upper bound (implicit lower bound = 1). This upper bound expresses the number of elements in the vector; the element size will be defined by the instruction using the vector and the type of its reference.

An N-dimensional array is a tuple of words consisting of N non-pointer triples (lower bound, stride, upper bound) followed by a reference.

2 Program in PerqFlex

While running program, Flex is always obeying the code of some procedure, the current procedure.

2.1 The local stack

The locals of this procedure,the current locals, are contained in a work-space block (type 1); these locals are directly accessible using instructions such as load_1_word_local (op code 0). Two other areas are similarly accessible - the non-locals (eg op code 1) and the constants (op code 2) of the current procedure. The non-locals (if any) are in one of the blocks (type 4) pointed at in the closure which forms the current procedure while the other gives a type 2 block containing pointers to both the constants and code of the procedure.

The locals operate as a stack entirely contained within the current work-space block. The next free word on this stack is the stack-front (sf). Clearly sf is always constrained to lie within the limits of the current work-space; any attempt (either explicitly or implicitly) to go outside its bounds will result in an error.

2.1 The U register and tos

There is only one general purpose register in PerqFlex - the universal register U. U may hold:

any number of words
a single character or a single boolean
an illegal value (ie an Exception)

or void (ie the unique value requiring no bits)

The instructions which load U (op codes 0 - 37 etc) push the old value in U (provided it is not an exception) onto the local stack (updating sf in multiples of words) before loading the new value.

Most of the arithmetic instructions use a value on top of the stack (tos) together with the value on U to produce results. The tos value is removed and sf reduced by the operation of the instruction. The number of words in tos depends on the particular instruction, and also sometimes on the value in U (eg equality, op code 114). Thus the int_multiply instruction (op code 100) multiplies a 1-non-ptr U by a 1-non-ptr tos, giving the answer as a 1-non-ptr, reducing sf by 1 word. The real_multiply instruction works similarly removing 2 words from the stack while the equality instruction removes the number of words required to hold the value in U.

2.3 Program control

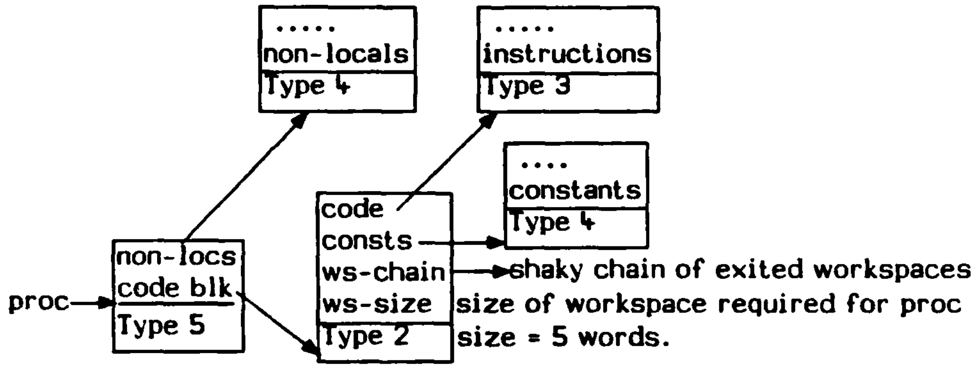
The flow of control instructions only allow jumps within the current procedure code and even then only in a restricted form in that usually only forward jumps are allowed to carry a non-void U. The only ways to escape from the current procedure code is to call another procedure, exit from the current one, fail, or obey the goto instruction (op code 71). The address of the instruction currently being obeyed is held in the program control register pc; clearly pc is constrained to be within the limits of instructions within the current codeblock.

2.4 Privilege

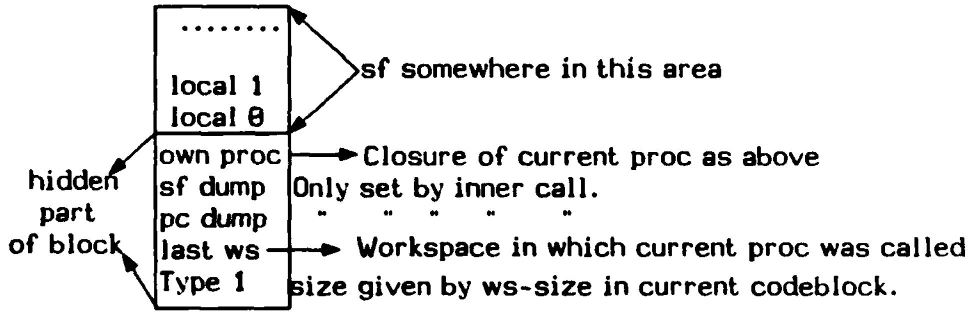
Some of the instructions are only allowed if the procedure code is being run in privileged state. These instructions are mainly concerned with peripheral transfers. This state is a property of the procedure itself. If one calls a locked procedure then its code will run in privileged state otherwise in unprivileged state. Thus, if an unprivileged procedure is called from within a privileged one, on exit from the inner call the outer will remain privileged and similarly in the reverse sense. Clearly the operation which locks a procedure can only be obeyed in privileged state.

3 Procedures

A procedure is a pointer to a closure block of size 3 words:



The current work-space block is:


....
local 1
local 0
sf somewhere in this area
own proc
sf dump
pc dump
last ws
Type 1
Closure of current proc as above
Only set by inner call.
" " " " "
Workspace in which current proc was called
size given by ws-size in current codeblock.
hidden part of block

The workspace chain given by the last ws chain in a work-space is terminated by a zero word, in the first work-space of a process.

3.1 Procedure calls

The action of a procedure call (op codes 64-67) is as follows. The current values of pc and sf relative to their respective blocks are stored in the second and third words of the current workspace. The privilege state and the TD state (see 4) also stored along with sf. The codeblock derived from the procedure to be called is now examined. It contains, in its first two words, information to produce a work-space block for the procedure being called. If the second word (ws-chain in diagram) is a pointer, then we know (see exit 3.2) that this is shaky pointer to a chain of work-spaces suitable for running this procedure and hence we have the desired work-space by removing it from this chain. If ws-chain is not a pointer, then a new work-space block is generated given its size in the first word in the standard manner. This may involve one in a garbage collection so the procedure call instructions are arranged so that they can be restarted after such a garbage collection.

Having produced a work-space suitable for the new procedure, a pointer to the current work-space is put in its first word, and the new procedure in its fourth. This new work-space now becomes current, sf is set to its fifth word (word 0 of locals), pc to the first byte of the instruction block pointed to by the fourth word of the new codeblock, and the internal registers set up so that the current locals, non-locals and constants come from the new locals, non-local block and constant respectively. If the procedure being called is locked, the code will run in privileged state; otherwise it is unprivileged.

During all of this, the contents of U remain unchanged so that parameters to a procedure are normally passed in U.

3.2 Exit from procedures

Exit from a procedure (op codes 68, 69) is essentially the reverse process: if a pointer to the current work-space has not been loaded while obeying the current procedure (ie instructions 34 or 168 have not been encountered) then the current work-space is put on the ws-chain in the current codeblock. The previous work-space (in 1st word of current work-space) is now made current, and sf, pc and the two states are reset from their dump positions in this work-space. The locals are made current in this work-space and non-locals and constants are reset from the own-proc dumped within it.

During exit, the contents of U remain unchanged so that results of procedures are normally passed in U.

3.3 Demand loading

The description of procedure call given above is somewhat simplified in that the codeblock or the entire procedure can reside on filestore. In both cases, the appropriate pointers will be filestore capabilities represented by pointers to keyed blocks (type 6). The firmware will interpolate a call of a system procedure, load_proc, in the operation of a call instruction whose closure is, or contains, such a keyed block. Load_proc is rather similar to the scavenge procedure, in that it can accept any parameter. It can access the keyed blocks as non-locals, and it uses information from the keyed blocks to load the actual code etc, from backing store into mainstore. Load_proc will then exit to repeat the original call instruction that provoked it, ensuring that this kind of closure is only fully loaded when it is actually called. The alias fields of the filestore capabilities will ensure that the discs are not accessed if the procedure or codeblock is already in mainstore.

4 Exceptions

An exception in Flex occurs when some attempt is made to break the rules of the Flex instruction set. All exceptions have a characteristic word-pair associated with them - those raised directly by the micro code consist of zero followed by a small integer.

The treatment of an exception depends on whether it was raised in one of two states either T-state or D-state. This state can be set by instructions (op code 94 and 95) and is carried into inner procedure calls. On exit from a procedure, the T or D state is set to what it was on entry.

4.1 Errors and failures

Exceptions arise in two slightly different ways, called, for want of better words, errors and failures.

An error occurs where any attempt to continue with the instruction would compromise the access rules for blocks and pointers. Roughly speaking,one could say that they are the compiler's fault. They include attempts to access outside the limits of a block or applying the wrong type of operands to instructions.

A failure tends to be more data-dependent and more likely to be the program writer's fault. Typical failures are arithmetic overflow and indices out of bounds. Also included are the explicit exceptions raised by the fail and exit_fail instructions (op codes 173, 69) where the characteristic is given by the operand U.

The only difference between failures and errors occur when the exception is raised in T-state. In this case, a failure produces an illegal value in U which has an associated word-pair identical to the characteristic of the exception. Any attempt to use an illegal value in instructions other than those explicitly designed to deal with them (op codes 92, 165 etc) will result in an error whose characteristic is the same as that associated with the illegal value. This implies that one can deal with failures like overflow in the current procedure by using these illegal-handling instructions.

In T-state, an error results in the premature exit from the current procedure with an illegal value in U whose word-pair is the characteristic of the error. Since illegal values give errors unless explicitly tested for, the net effect is that an entire chain of procedure calls are exited from until one is encountered which is prepared to accept an illegal result.

In D-state (Diagnostic state) all exceptions are treated identically. In essence, a system procedure, fail_proc, is called in place of the current procedure, so that, if an exit was obeyed in fail_proc, it would exit to the same place as the current procedure. The parameters of the call of fail_proc give access to the locals and codeblock of the failing procedure and the characteristic of the exception is available as a non-local. The (softwared) action of fail_proc is to construct a chain of failing environments. It does this by constructing an element of the chain and then doing an exit_fail with a reference to the element in U. Eventually, some lower procedure will gather up the resulting illegal and use it to construct visible diagnostics for the exception. As far as the firmware is concerned, all that it does at an exception in D-state is to find fail_proc in system_block, dump the characteristic into system_block, and call fail_proc with a four word parameter consisting of a pointer to the current locals, relative values of pc and sf, and a pointer to the local codeblock.

5 Storage allocation

Only the micro-code regards Flex main store as a linear store addressable from end to end. The macro-code which is the Flex instruction set only understands blocks and pointers to them, so that Flex program can only address those disjoint unrelated blocks for which it has pointers of the right sort. Running programs will involve new blocks being created to hold data, for example by using the generate instructions (72-74 etc) or simply by calling a procedure which requires a new work-space block. Thus the micro-code which implements those instructions simply grabs a new empty block from the top of a continually growing stack in the linear store, putting in the appropriate overhead word and delivering a pointer as result.

This linearly growing stack will eventually encompass the entire physical store and at this stage garbage collection occurs. The garbage collector notes all of blocks which are still "live", and compacts all live blocks down to the bottom of store, updating all pointers in them so that they still point to the same data. Thus the space occupied by "dead" blocks is recovered and, hopefully, there will be sufficient room in the linear store for the request for a new block which provoked the garbage collection.

Clearly the address actually held in a pointer can change on each garbage collection. However since all pointers to a given block are changed consistently and since arithmetic is forbidden on pointers, one can regard pointers as immutable objects in Flex programs.

A live block is either a unique block, system_block, or else is pointed to from within another live block. System_block is a block known to the micro code and contains the interrupt procedures and other goodies to keep alive all currently active processes.

The actual sequence of events which happens in the micro code at a garbage collection is as follows. The micro code discovers that a request for the generation of a block cannot be satisfied from the linear store. It then interpolates the call of a procedure, scavenge, before the current instruction. Scavenge (which is found in system block) is a peculiar procedure in that one can guarantee that there will always be a workspace available for it and that it can accept any value in U as a parameter. This last is necessary since the instruction requiring the block could be a procedure call which is meant to leave the value of U unchanged. The privileged scavenge procedure dumps the value of U (op code 206), obeys the garbage_collect instruction (op code 204), and then finds if the current store demand can be satisfied; if it can then the dumped value of U is reinstated (op code 205) and scavenge is exited - to repeat the store grabbing instruction. If it cannot be satisfied, then some process must be failed so that store can be released to continue.

6 Interrupts

Time critical interrupts (to deal with the screen refresh, for example) are serviced entirely in micro-code without disturbing the Flex procedure and process structure. Some of these, of course, may require attention at the Flex level and this attention will be initiated by a Flex-interrupt at the next available opportunity. As an example, the micro-code services both a real-time clock and an interval timer every sixtieth of a second, only interacting at the Flex level when the timer count expires (see instructions with op codes 36 and 216 for reading and setting this counter).

A Flex-interrupt can only occur when U contains void and the instructions are being obeyed in non-privileged state. This sometimes occurs in the middle of an instruction where repeating the instruction would do no harm. This is the case in the load instructions where U is void after it has been pushed but before the new value has been loaded into U; since pushing void is a null operation, the instruction has the same effect whether or not it has been interrupted and restarted.

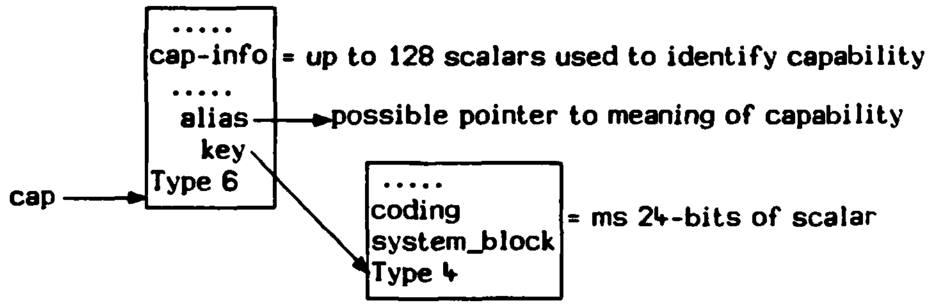
When a Flex-interrupt occurs, the effect is exactly the same as if a parameterless procedure (delivering void) had been called in the code being interrupted. This procedure depends on the type of interrupt and is found in system_block. The calling sequence of an interrupt procedure will not invoke garbage collection (although inner calls may) and is intended to be run in privileged state.

Sixteen different Flex-interrupts are possible, each with its own procedure in system_block. They are arranged to be called in priority order, each interrupt being queued until it is the one of highest priority.

7 External capabilities

Capabilities which are in some sense external to the mainstore of Flex are represented in Flex by locked pointers to type 6 keyed blocks. As an example, filestore capabilities are "pointers" to data on disc. The contents of the type 6 block allows one to determine where on the disc. The key of the type 6 block is a pointer which determines the particular filestore. In general, the key determines how (or whether) one can transput these external capabilities, together with a system-wide coding which gives its identity. Thus although one can store filestore capabilities themselves on filestore, cross filestore capabilities (those with different codings) are in general not allowed; one cannot point from one filestore to another. This is reflected in mainstore by the fact that the keys of capabilities to different filestores have different keys. Note that since the keys are pointers , they are unforgeable and, hence, only procedures which know the key can access the information in the block.

The detailed format of the representation of an external capability in mainstore is:


.....
cap-info
.....
alias
key
Type 6
cap
= up to 128 scalars used to identify capability
possible pointer to meaning of capability
.....
coding
system_block
Type 4
= ms 24-bits of scalar

The pointer to the system_block in the first word of the key is there to ensure that only authorised procedures can create external capabilities; ie those who can access system_block.

The alias word in the keyed block is usually used by the access procedures as a short cut; for example, after reading the data corresponding to a filestore capability into a block of mainstore, the alias field will contain a shaky pointer to that block to avoid unnecessary later reads of the disc. In order to make this effective, the instructions which receive external capabilities from the outside world will ensure that there is at most one copy of the type 6 block with the same cap-info and key in mainstore at any one time.

Just as in mainstore, it is necessary to be able to distinguish capabilities from scalar data on the external media, be it in networks, or in filestore etc. Flex filestore, for example, uses blocks in filestore and pointers to them in much the same way as in mainstore. Just as in mainstore, some of the filestore blocks cannot contain capabilities, in analogy to the type 3 and 11 blocks in mainstore.

However, the majority of filestore blocks are analagous to type 2 (codeblock),4 (normal) or 5 (procedure) blocks which will contain other capabilities usually to other data in the same filestore. In these instances, data in transferred in multiples of 32-bit words where, just as in mainstore, the least significant bit differentiates between scalars and capabilities.

A privileged instruction (op code 225) transforms the mainstore representation of an external capability to a form suitable for sending to external devices. This external representation comes in either the long form or the short form. If the key of the capability is the same as the one given as another operand to the instruction then the short form is produced otherwise it is the long form which includes the coding in the capability key:

Long form:

Word 1	:	24-bit coding in key in Type 6	1	6-bit size	1
Word 2	:	....			
....		copy of cap-info in Type 6 block			
Word $_{size}$	:	....			

The short form is just the first two words of the cap-info in the type 6 block, with the least significant bit of the first word set with its eightth bits clear.

The instruction (opcode 233) is also privileged, and produces a mainstore representation of an external capability from the cap-info in these external representations. It uses a key derived from the coding in the first word of the long form and from some procedural context in the short form. For example, a filestore capability to a Winchester disc on the disc itself is expressed in the short form. This instruction makes use of a 256-entry hash table, held in system_block, to search for an already existing copy of the capability in mainstore so that there are never duplicates of an external capability in mainstore.

These transformations are done via a buffer area of main store which is used for general peripheral transfers, including dma transfers. It is unaffected by garbage collection and is the area referred to in the buffer transfer instructions.

8 PerqFlex instruction set

Instructions are usually 1, 2 or 3 bytes long, the first defining the operation code. The remaining bytes, if any, are denoted by a & sz (1 byte quantities) or p (two byte quantity). The a-field generally is a data displacement and the sz-field gives data size. The a and sz fields can be effectively extended by using the modify_next instruction (op code 76). A p-field is a byte displacement from the beginning of the current procedure code.

Almost every instruction can cause exceptions. These can arise in many different ways eg the operand(s) of the instruction are of the wrong type for the instruction or displacements given are too big for the blocks. The exceptions raised by breaking the rules in such a way will have a characteristic pair consisting of (0, small integer) (see 9.6).

Due to an idiosyncracy of the Perq hardware, instruction bytes are expressed in a different order to that implied by the character indexing and dereference. The first instruction byte is in the least significant half of each 16-bit word while indexing works with the most significant half first. The procedures for constructing codeblocks on filestore regularise this position by re-ordering the bytes of the string giving the instructions of the codeblock.

A * superscript in the description of an instruction means that the instruction is Flex-interruptable at that point.

8.1 Unprivileged instructions

Load_1_word

0 , a	: Push U * , U:= a th word of locals.
1 , a	: Push U * , U:= a th word of non-locals.
2 , a	: Push U * , U:= a th word of constants.
3 , a	: U:= a th word of block pointed at by U.

Load_2_words

4 , a	: Push U * , U:= word-pair at a th word of locals.
5 , a	: Push U * , U:= word-pair at a th word of non_locals.
6 , a	: Push U * , U:= word-pair at a th word of constants.
7 , a	: U:= word-pair at a th word of block pointed at by U.

Load_N_words

8 , a , sz	: Push U^* , U:= sz+3 words at a^{th} word of locals.
9 , a , sz	: Push U^* , U:= sz+3 words at a-th word of non-locals.
10, a , sz	: Push U^* , U:= sz+3 words at a-th word of constants.
11, a , sz	: U:= sz+3 words at a^{th} word of block pointed at by U.

Load_1_character

12, a	: Push U^* , U:= character in a^{th} word of locals.
13, a	: Push U^* , U:= character in a^{th} word of non-locals.
14, a	: Push U^* , U:= character in a^{th} word of constants.
15, a	: U:= character in a^{th} word of block pointed at by U.

Load_1_boolean

16, a	: Push U^* , U:= bool in a^{th} word of locals.
17, a	: Push U^* , U:= bool in a^{th} word of non-locals.
18, a	: Push U^* , U:= bool in a^{th} word of constants.
19, a	: U:= bool in a^{th} word of block pointed at by U.

Load_ptr_to_current_areas

28	: Push U^* , U:= locked ptr to non-locals block.
29	: Push U^* , U:= locked ptr to constants block.

Load_literal

30, a	: Push U^* , U:= a (Char).
31, a	: Push U^* , U:= a (Bool)
32, a	: Push U^* , U:= a (Int).
33	: Push U^* , U:= void.

Load_ptr_to_locals

34	: Push U^* , U:= ptr to locals block.

Load_times

35	: Push U * , U:= time of day (Int jiffies).
36	: Push U * , U:= unexpired slot-time (Int jiffies). 1 jiffy = 1/60th second

Push_and_take

37, sz	: Push U * , U:= sz words on tos.

Store_U

40 , a	: Stores U (unexceptional) at a th word of locals ,U:=void *.
43 , a	: Stores U (unexceptional) at a th word of block pointed at by tos, U:= void*.

Select_from_U

44 , a , sz	: U:= sz words starting at a th word of U.

Date

45	: Push U * , U:= Int date as days after 31st Dec 1982 { assuming 31 days per month}

Shift

46	: U:= tos Leftshift U; (Int,Int) → (Int,Int) ie if tos = $X*2^{31-U}$ + Y then U := $(X,Y*2^U)$

Select_ref

47 , a	: Select ref i.e. add a to last word of U.

Deref

48 , a , sz	: Deref word vector in U i.e. U:=(UPB vector * a + sz) words pointed at by vector.
49 , sz	: Deref word ref in U i.e. U:= sz words pointed at by ref.
51 , sz	: Deref char ref in U; NB sz ignored U:= character pointed at by ref.
53 , sz	: Deref bool ref in U; NB sz ignored U:= boolean pointed at by ref.

Pack and Unpack

54	: Unpack i.e. U:= word contents of block pointed at by U. Unpack with U = Exception is null instruction
55	: Pack i.e. U:= ptr to block (type 4) containing copy of U.

Vector Operations

56 , sz : Index vector in U with element size sz by index on tos giving ref to element in U;
Given U = (b,p,d) and tos = i,
U:=(p,d+(i-1)*sz) where $1 \le i \le b$.

57 , sz : Trim vector on tos with element size sz by Int pairin U giving trimmed vector in U;
Given U = (i,j) and tos = (b,p,d),
U:= (jm-im+1,p,d+(im-1)*sz)
where im= max(1,i) and jm = min(b,j).

58 : If UPB vector on tos $\neq$ UPB vector in U then fail (0,2).

Array operations

59 , a : Index a+1 dimensional array in U by a+1 indices on tos giving ref to element in U;
Given U = $(lb_0,s_0,ub_0, \ldots,lb_a,s_a,ub_a, p,d)$
and tos = $(i_0,i_1,\ldots, i_a)$,
U:=$(p,d + s_0*(i_0-lb_0) + \ldots + s_a*(i_a-lb_a))$
where $lb_n \le i_n \le ub_n$ for $0 \le n \le a$.

60 , a : Trim a+1 dimensional array on tos by integer triple in U giving a+1 dimensional array in U;
Given U = (fu,tu,nlb)
and tos = $(lb_0,s_0,ub_0, \ldots,lb_a,s_a,ub_a,p,d)$,
U:=$(nlb,s_0,nlb+t-f, lb_1,s_1,ub_1,\ldots, lb_a,s_a,ub_a,$
$p, d+(f-lb_0)*s_0)$
where $f = \max(lb_0,fu)$ and $t = \min(ub_0,tu)$.

61 , a : Slice a+1 dimensional array on tos by index in U, giving a dimensional array in U.
Given U = i
and tos = $(lb_0,s_0,ub_0, \ldots,lb_a,s_a,ub_a,p,d)$,
U:= $(lb_1,s_1,ub_1, \ldots, lb_a,s_a,ub_a,p, d+ s_0*(i_0-lb_0))$
where $lb_0 \le i \le ub_0$

Unite

62 , a , sz : Unite U with a and make it a sz word object, i.e.
U:= (a, U, 0,...).

Assign

63	: Assign (unexceptional) U to position given by ref on tos, and let U := ref;

Procedure calls and exits

64 , a	: Call the procedure given at a th word of locals.
65 , a	: Call the procedure given at a th word of non-locals.
66 , a	: Call the procedure given at a th word of constants.
67	: Call the procedure given on tos.
68	: Exit from current procedure.
69	: Exit from current procedure and fail U .

Goto

71	: Goto label given in U, where label is pair (pointer to destination workspace, p-displacement in codeblock).

Generate new blocks

72	: U:= ptr to new closure (type 5) formed from ptr to code (type 2 or 6) in U and ptr to non-locals (type 4) or zero on tos; (Ptr,Word) → Proc.
73	: U:= ptr to new normal array block (type 4) of size in words given by U.
74	: U:= ptr to new block (type 3) of size in words given by U.

Modify_next

76 ,a1,sz1	: Modify the a & sz fields of next instruction (if present) by a1*256 & sz1*256.

Stack front operations

77 , a	: Set sf to start of locals + a words.
78 , a	: If sf ≠ start of locals + a then fail (0,5)

Discard

79	: U:=void *.

Operations on pointers

80	: U:= shake U; Ptr → Ptr.
81	: U:= firm U; Word → Word; Scalars unchanged.
82	: U:= U is a ptr; Word → Bool.
83	: U:= block type of ptr in U; Ptr → Int.
84	: U:= byte block size of ptr in U; Ptr → Int.

Null instructions

85	: Null instruction
86	: Null instruction

Modify_next_dynamically

87	: Modify the a & sz fields of next instruction by int pair on tos.

Jumps and branches

88 , p	: IF U then jump to p FI , U:= void *.
89 , p	: IF not U then jump to p FI , U:= void*.
90 , p	: IF U then jump forward to p ELSE U:= void* FI.
91 , p	: IF not U then jump forward to p ELSE U:= void * FI.
92 , p	: IF U is illegal then jump to p, U:=void * FI.
93 , p	: Jump to p (if U not void then jump must be forward).

Set failure state

94	: Set D-state.
95	: Set T-state.

FOR instructions

```
96 , p   : For test; (FOR,BY) in U , TO on stack ;
           IF (TO-FOR)*BY < 0
              THEN U:=void, Pull TO , jump to p* FI.
97 , p   : For step; * (FOR,BY) on tos , U:=(FOR+BY,BY), jump to p.
```

Switches

98 , a : Case switch; jump to next + (1 ≤ U ≤ a | 3*U | 0).

99 : Associative switch - tests equality or ranges;
Followed by sequence of byte triples or quads:
(b_i, p_i) or (x_i+128, y_i, p_i)
terminated by $(b_n=0, p_n)$, where $b_i<128$ and $x_i<128$;

```
FOR i DO
 IF U=$b_i$ (single byte) OR U ≥ $x_i$ AND U ≤ $y_i$ THEN
  jump to $p_i$, U:=void*
 ELIF $b_i$ = 0 THEN jump forward to $p_i$ FI
OD.
```

Integer arithmetic

100	: U:=tos+U; (Int,Int)→Int.
101	: U:=tos-U; (Int,Int)→Int.
102	: U:=tos*U; (Int,Int)→Int.
103	: U:=(remainder,tos/U); (Int,Int)→(Int,Int); Sign of non-zero remainder is same as divisor.

Integer tests

104	: U:=tos ≥ U; (Int,Int)→ Bool.
105	: U:=tos < U; (Int,Int)→ Bool.
106	: U:=tos ≤ U; (Int,Int)→ Bool.
107	: U:=tos > U; (Int,Int)→ Bool.

Monadic operations

108	: U:= ABS U; Int → Int.
109	: U:= -U; Int → Int.
110	: U:= ABS U; (Char or Bool or Word) → Int.
112	: U:= REPR U; Int → Char.
113	: U:= ODD U; Int → Bool.

Equality

114	: U:= tos=U (Any,Any) → Bool.
115	: U:= tos≠U (Any,Any) → Bool.

Logical Operations

116	: U:= tos OR U; (Bool,Bool)→Bool or (Int,Int)->Int.
117	: U:= tos AND U; (Bool,Bool)→Bool or (Int,Int)->Int.
118	: U:= tos EXOR U; (Bool,Bool)→Bool or (Int,Int)->Int.
119	: U:= tos EQUIV U; (Bool,Bool)→Bool or (Int,Int)->Int.
121	: U:= NOT U; Bool → Bool or Int→Int.

String equality

122	: U:=(string in vector on tos = string in vector in U); (Vec Char,Vec Char) → Bool.
123	: U:=(string in vector on tos ≠ string in vector in U); (Vec Char,Vec Char) → Bool.

Real arithmetic

124	: U:= tos+U; (Real,Real)→Real or (ShortReal,ShortReal)→ShortReal.
125	: U:= tos-U; (Real,Real)→Real or (ShortReal,ShortReal)→ShortReal.
126	: U:= tos*U; (Real,Real)→Real or (ShortReal,ShortReal)→ShortReal.
127	: U:= tos/U; (Real,Real)→Real or (ShortReal,ShortReal)→ShortReal.

Real tests

128	: U:= tos ≥ U; (Real,Real) or (ShortReal,ShortReal) → Bool.
129	: U:= tos < U; (Real,Real) or (ShortReal,ShortReal) → Bool.
130	: U:= tos ≤ U; (Real,Real) or (ShortReal,ShortReal) → Bool.
131	: U:= tos > U; (Real,Real) or (ShortReal,ShortReal) → Bool.

Real monadic operations

132	: U:= ABS U; Real→Real or ShortReal → ShortReal.
133	: U:= - U; Real→Real or ShortReal → ShortReal.
134	: U:=ENTIER U; Real or ShortReal → Int.
135	: U:=ROUND U; Real or ShortReal → Int.
136	: U:= widen U; Int → ShortReal.
137	: U:= widen U; Int or LongInt → Real.
138	: U:= ENTIER U; Real → LongInt.
139	: U:= ROUND U; Real → LongInt.

Long arithmetic

140	: U:=tos+U ; (Int,Int or LongInt) → LongInt.
141	: U:=tos-U ; (Int,Int or LongInt) → LongInt.
142	: U:=tos*U ; (Int,Int) → LongInt.
143	: U:=(remainder,tos/U); (LongInt,Int) → (Int,Int); Sign of non_zero remainder is same as divisor.

Long to decimal

144	: U:=(remainder,U/10); LongInt → (Int,LongInt).

Long conversions

145	: U:= LENGTHEN U; Int → LongInt.
146	: U:= SHORTEN U; LongInt → Int.

Decimal to long

147	: U:= U*10 +(U ≥ 0 | tos | -tos); (Int,LongInt) → LongInt

Long tests

148	: U:= tos ≥ U; (LongInt,LongInt) → Bool.
149	: U:= tos < U; (LongInt,LongInt) → Bool.
150	: U:= tos ≤ U; (LongInt,LongInt) → Bool.
151	: U:= tos > U; (LongInt,LongInt) → Bool.

Real conversion

152	: U:= LENGTHEN U; ShortReal → Real.

Max and Min

154	: U:= max(tos,U); (Int,Int) → Int.
155	: U:= min(tos,U); (Int,Int) → Int.

Range checks

156, a , sz	: U:= (a ≤ U ≤ sz); (Int or Char)→ Bool.
157	: Given U = (l,u), U:= (l ≤ tos ≤ u); (Int or Char,(Int,Int)) → Bool.

Keyed block operations

158	: U:= open ptr to new keyed block of size U words; Int → Ptr.
159	: U:= locked version of pointer in U; Ptr → Ptr or Ref → Ref.
160	: U:= open ptr to keyed block in U with key on tos; (Word,Ptr or Ref) → Ptr or Ref.
161	: Given U = pointer to keyed block with system block as key, U:= 2nd word of block. Ptr → Word

Load d_to_b

162	: Push U * , U:= d_to_b (in 9th word of system block).

Decimal exponent conversions

163	: U:= (e,l) given $l*10^{e} = U$; ShortReal → (Int,Int) or Real → (Int,LongInt).
164	: U:= $U*10^{tos}$; (Int,Int) → ShortReal or (Int,LongInt) → Real.

Unite with Exception

```
165, sz    : U:=IF  U isnt exception THEN  (1,U,..,0,...)
                 ELSE (2,characteristic word pair of U,..0..,)
                 FI;
             (Any or Exception) → sz*Word.
```

Vector pack and unpack

166	: Pack U into new vector; (n*Word) → Vec Word, Char → Vec Char or Bool → Vec Bool.
167	: Unpack vector in U to produce n*words, char or bool in U; Vec Word → n*Word, Vec Char → Char, or Vec Bool → Bool. Unpack applied to an exception is a null instruction.
172	: Null instruction

Fail

173	: U:= Exception formed from word-pair in U; (Word,Word) → Exception.
173 , a	: U := IF U THEN void ELSE Exception(0,a) FI; Bool → Exception

Generate Char, Bool and Code blocks

174	: U := pointer to new type 3 block just large enough to contain U chars; Int → Ptr.
175	: U := pointer to new type 11 block just large enough to contain U bools; Int → Ptr.
176	: U := pointer to new code-block (type 2) given by U = (ws, nls, instr) where: ws = no of words of locals required by codeblock nls = ptr to type 4 block containing constants instr = ptr to type 3 block containing instructions (Int,Ptr,Ptr)→Ptr.

Char and Bool multiple assignments

180	: Assign chars in Vector given by U to Vector given on tos and let U = Vector from tos.
181	: Assign bools in Vector given by U to Vector given on tos and let U = Vector from tos.
182	: Assign bools in 2-d array given by U to 2-d array given in on tos and leave U unchanged. 1st stride of both arrays must be a multiple64 2nd stride of both arrays must be 1.

Real conversion

183	: U:=SHORTEN U; Real → ShortReal.

Load literal string

184 , sz : Push U * ; U := (sz,instr,pc+2) and pc:=sz+2+d
where pc is the current program displacement and is odd before the instruction and d =0 or 1 is chosen to make it odd after,
and instr is locked version of current instruction block.

Bool array operations

184+i : f_i(Bool 2-d array on tos, Bool 2-d array in U), $1 \le i \le 7$
U unchanged;
1st stride of both arrays must be a multiple of 64
2nd stride of both arrays must be 1.
Where

$f_1(a,b)$ = a:= NOT b,
$f_2(a,b)$ = a:= a AND b,
$f_3(a,b)$ = a:= a AND NOT b,
$f_4(a,b)$ = a:= a OR b,
$f_5(a,b)$ = a:= a OR NOT b,
$f_6(a,b)$ = a:= a EXOR b,
$f_7(a,b)$ = a:= a EXOR NOT b.

Multiple adds and subtracts

192 : U:=tos+U ; (N*Int,N*Int) → N*Int
ie (1,2) + (4,5) = (5,7)
193 : U:=tos-U ; (N*Int,N*Int) → N*Int

Enable

201 : Set non-privileged state.

8.2 Privileged instructions

Any attempt to use these instructions in unprivileged mode will result in error(0,11).

Append

153	: U := tos Append U ,where tos = ref to block whose 1st word is chain ending in zero. (Ref,Word or Ref) → Ref

Lock procedure

159	: U := locked version of proc pointer in U

Winchester disc

194, 0	: Set disc header address from U (Vec Char); U:=void.
194, 1	: Set disc data address from U (Vec Char); U:=void.
194, 2	: Start disc access. U = (Int action, Int disc_address); U:=void.
194, 3	: U := disc status (Int).
194, 4	: U := disc_address (Int) obtained from next 4 bytes of currently selected i/o buffer.
194, 5	: Put disc_address (Int) in U into next 4 bytes of currently selected i/o buffer; U:=void.
194, 6	: U := outermost disc_address (INT)
194, 7	: U := ordered sequence number of disc_address in U; Int → Int.
194, 8	: Read disc drive details from i/o buffer.
194, 9	: tos < U in disc address sequence; (Int,Int) → Bool.

Laser printer

194, 16	: Set left and right hand margins to l=16 and r=16 pixels and set get ready state where U = (l,r); U:= status of printer; IntPair → IntPair.
194, 17	: If s = 1 then start printer and insert l pixel lines, where U=(s,l); IntPair → Void.
194, 18	: If U = 0 then U:=status of printer Elsf U = 1 then U:=status at last interrupt Elsf U = 2 then Reset interface; U:=(0,0) Fi; Int → IntPair.
194, 19	: Output l pixel lines starting from bit position given by r alligned to 16 bit word boundary where U = (l,r); (Int,Ref Bool) → Void.

Scavenge

204	: Do a garbage collection, delivering the number of words recovered in U as an Int.

Dump and reset U

206	: Dump U (in internal form) to the first 5 words of the current work-space, leaving U void.
207	: Reset U from dumped value in first 5 words of the current work-space.

Ethernet channel

208	: Ethernet accept; Vec Char → Void.
209	: Ethernet read; Vec Char → Void.
210	: Ethernet send; Vec Char → Void.

Z80 devices

212	: Push U* , U:= selected device interrupt reason (Int).
213	: Fast output data for selected device; Vec Char → Int; U := unsent size of previous data (Int).
214	: Selected device status input buffer; Vec Char → Int; U := pevious status buffer size.

Set times

215	: Set time (jiffies) from Int in U; U:=void. or date and time from (Int,Int) in U
216	: Set interval timer (jiffies) from Int in U; U:=void.

Load ref to system_block

217	: U:= Ref to Uth word of system_block; Int → Ref.

I/O buffers

219	; Write Int in U to next byte in buffer if there is room and deliver TRUE ; othewise FALSE . Int → Bool Bytes in Perq i/o order
220	; Select IO buffer; Vec Char → Void
221	; Write INT in U to next byte in buffer if there is room and deliver TRUE ; othewise FALSE . Int → Bool Bytes in character index order
222	; Write INT in U to next 2 bytes in buffer if there is room and deliver TRUE ; othewise FALSE . Int → Bool
225	; If U is an external capability, and the first word in its block is identical to the word on tos, then the short form of its representation is written to the buffer; Other capabilities are written in long form. If U is not a pointer, then the word on tos is irrelevant and the bytes written are an (even) byte representation of the integer in U. U := (room in buffer) (Word,Word) → Bool.
226	; Set buffer index to U (Int); U:=void
227	; Push U ; U := next byte from buffer as Int. Bytes in Perq i/o order
228	; Push U ; U := next buffer index(Int).
229	; Push U ; U := next byte from buffer as Int. Bytes in character index order
230	; Push U ; U := next 2 bytes from buffer as Int.
231	; Push U ; U := next 4 bytes from buffer as integer.
233	; U := next capability in buffer, given key in U Ptr → Word
235	; Push U; U := type of next word in buffer

Special Areas

242	; U := ptr to the block (type 11) containing the default character font.
243	; U := ptr to the block (type 3) containing all I/0 buffers.
244	; U := ptr to the block (type 11) containing the bits for the screen raster.

Make and break blocks

239	: U := word-pair in first two words of block pointed at by U; Ptr → (Word,Word)
240	: Assign word pair in U to block pointed at by tos: (Ptr,(Word,Word)) → void.

Z80 devices

245	: Control data for selected device depending on U type: Vec Char: set new input buffer; U := unused size of previous buffer. Int: Increase unused part of input buffer (wrap-round); U := previous unused size. (Ref Char,Int,Int,Int): Implement input flow control; U := void.
246	: Select Z80 device from Int in U; U:=void.
247	: Send message from Vec Char in U to Z80; U:=void.

Screen pointer

248	: Assign screen/pointer combination function (Int) and start address of cursor pointer from U; U:=void (Int, Ref Char) → void
249	: Set vertical position of pointer from U; Int → Void
250	: Set horizontal position of pointer from U; Int → Void

High resolution tablet

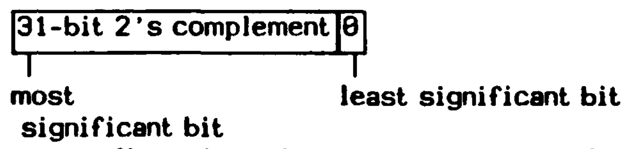
251	: If U then GPIB input to be interpreted as tablet input; Bool → Void

9 Data and block formats

9.1 Data Objects

Integers

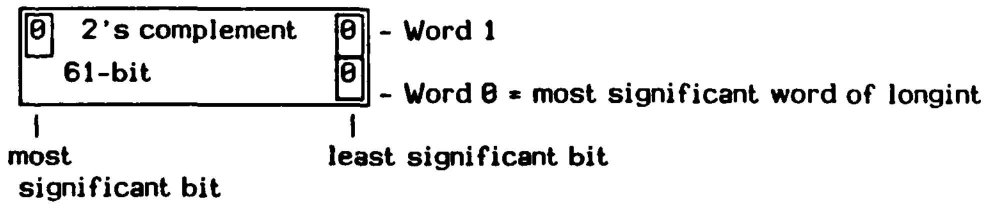
One word:



Any overflows in an integer operation produces the exception fail(0,3) as the result of the operation.

Long integers

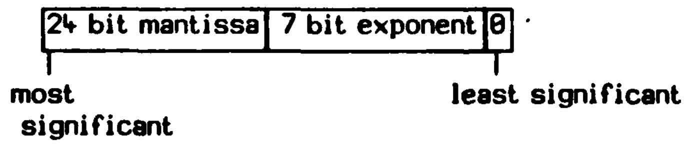
Two words:



Any overflows in a long integer operation produces the exception fail(0,3) as the result of the operation. The least significant 30 bits of a long integer are held as a positive integer to allow easy extension to multiple length arithmetic.

Short Reals

One word:



The mantissa of a short real is actually a 25 bit normalised 2's complement fraction. Since the sign bit of such a non-zero fraction is always the complement of its most significant bit, the sign bit is not stored. Short real 0.0 is held specially as all zero bits. The exponent is biassed by 64 so that the true binary exponent is 64 less than the 7-bit pattern given in the exponent field. Any overflow in an short real operation produces the exception fail(0,16) as the result of the operation.

Reals

Two words:

Word 1:	31 bit mantissa extension (ls part)			0
Word 0:	16 bit mantissa	0000	11 bits biased exponent	0

most significant — least significant

The mantissa of a real is actually a 48 bit normalised 2's complement fraction. Since the sign bit of such a non-zero fraction is always the complement of its most significant bit, the sign bit is not stored. Real 0.0 is held specially as all zero bits. The exponent is biassed by 1024 so that the true binary exponent is 1024 less than the 11-bit pattern given in the exponent field. Any overflow in a real operation produces the exception fail(0,16).

Pointers

One word:

		H-bit				Word address	
L	2 bit gc	0	4 bit type	S	3 bit spare	19 bit block address	1

1 → pointer locked
0 → pointer unlocked

1 → pointer shaky
0 → pointer firm

The type is repeated in block overhead word in same position with H-bit = 1.

References

Two words:

Bit disp if type = 11, char disp if type = 3 else word disp							0	:Word 1
L	2 bit gc	0	4 bit type	S	3 bit spare	19 bit block address	1	:Word 0

Vectors

Three words:

Bit disp if type = 11, char disp if type = 3 else word disp							0	: Word 2
L	2 bit gc	0	4 bit type	S	3 bit spare	19 bit block address	1	: Word 1
Upper bound of vector - lower bound = 1							0	: Word 0

The assignment, trimming and equality of vectors behave sensibly with zero sized vectors.

Arrays

3n+2 Words gives an n-dimensional array:

Bit disp if type = 11, char disp if type = 3 else word disp							0	Word 3n+1
L	2 bit gc	0	4 bit type	S	3 bit spare	19 bit block address	1	Word 3n
Upper bound 1 of array							0	Word 3n-1
Stride 1 of array							0	Word 3n-2
Lower bound 1 of array							0	Word 3n-3
....								......
Upper bound n of array							0	Word 2
Stride n of array							0	Word 1
Lower bound n of array							0	Word 0

The assignment, trimming and other operations on empty arrays behave sensibly.

9.2 Blocks

Workspaces

. etc .							X	
First local of proc							X	
L	2 bit gc	0	0101	0	3 bit spare	19 bit block address	1	: Proc pointer
P	000000000000	T	R	15 bit sf displacement			0	: Stack link
12 bit rubbish	4 bit ls link, bpc				15 bit ms link, apc		0	: Program link
0	2 bit gc	0	0001	0	3 bit spare	19 bit block address	1	: Last ws ptr
3 bit gc	1	0001	4 bit spare			19 bit word size	1	: Overhead word

The program and stack links are only stored at internal procedure calls and refer to the positions in the current procedure so that this proc can be restarted from this workspace. The various state bits in the stack link are:

R = 1 ⇒ current ws has been loaded.
T = 1 ⇒ current code runs in T-state.
P = 1 ⇒ current code runs in privileged state.

The sf displacement is a word displacement from the first local of the procedure.

The program link gives a codification of a displacement from the first byte of the instruction block pointed to by the codebock given by the proc pointer. The formula for getting the actual byte displacement is:

$apc*4+bpc-8$

Clearly the first workspace of a process can have no predecessor and hence its first word is zero.

Codeblocks

L	2 bit gc	0	0011	0	3 bit spare	19 bit block address	1	:Instructions
L	2 bit gc	0	0100	0	3 bit spare	19 bit block address	1	:Constants
0	2 bit gc	0	0001	1	3 bit spare	19 bit block address	1	:Shaky ws / zero
size of workspace required by code							0	
3 bit gc	1	0010	4 bit spare			19 bit word size+5	1	: Overhead word

If word 2 is a pointer it is a shaky one to a workspace suitable for this code block; this chain of shaky pointers is continued through word 1 of the workspace.

Character blocks

...... etc ....

8 Bit Byte 2	8 Bit Byte 3	8 Bit Byte 0	8 Bit Byte 1			
3 bit gc	1	0011	4 bit spare	19 bit word size	1	: Overhead word

The layout of characters here is the indexing order; this is consistent with the Vec Char assignments. Instructions, on the other hand, are read:

8 Bit Byte 3	8 Bit Byte 2	8 Bit Byte 1	8 Bit Byte 0
			least significant

Normal word Blocks

..... etc ......

pointer or scalar					X	
3 bit gc	1	0100	4 bit spare	19 bit word size	1	: Overhead word

Closures

L	2 bit gc	0	0100	0	3 bit spare	19 bit block address	1	:Non-locals/zero
L	2 bit gc	0	0010	0	3 bit spare	19 bit block address	1	:Codeblock
	3 bit gc	1	0101		4 bit spare	19 bit word size=3	1	: Overhead word

If there are no non-locals the non-word can be zero; the codeblock pointer may also be an external capability eg a codeblock on disc.

Keyed blocks

..... etc ......

pointer or scalar					X	
key					X	
3 bit gc	1	0110	4 bit spare	19 bit word size	1	: Overhead word

Boolean blocks

....etc .....

Q R S T U V W X Y Z .... A B C D E F G H I J K L M N O P						
3 bit gc	1	1011	4 bit spare	19 bit word size	1	: Overhead word

The bits are ordered as given by the alphabetical ordering given above starting from the most significant bit of each 16-bit word.

9.3 System_block, the screen map and buffer areas

There are three special blocks set up at system load. They are normal Flex blocks but their position in store means that they are not moved by garbage collection. The i/o buffer block is a character block whose pointer can be found with the privileged 243 instruction; slices of this will be used as buffers to peripherals. The screen block is a boolean block, usually accessed in program as a two dimensional array of bools; its pointer can accessed using the instruction 244.

System_block is a word block which will be the root of all accessible blocks at garbage collection. References to elements of system_block can be constructed using the privileged instruction 217.

System_block layout

Words 1 and 2	- Error words at failure; Word 1 also has proc during load_int and current work_space during scavenge.
Word 3	- Size in bytes of last demand for store during garbage collection.
Word 4	- Procedure invoked by failure; PROC(FAILQUAD) VOID fail_proc.
Word 5	- Procedure called when demand for space is not satisfied PROC(ANY)ANY scavenge.
Word 6	- Procedure called when calling a proc which is external capability . PROC VOID load_proc.
Word 7	- Procedure called when calling a proc with external capability as codeblock . PROC VOID load_codeblk.
Words 8 and 9	- Ref 256*Pair hash_table where Pair = (Shaky capability, (Ptr Pair or 0)) All external capabilities in mainstore are in hash_table.
Word 10	- Procedure to read external capability, d_to_b accessible by opcode 162.
Word 11	- Unused
Word 12	- Procedure called when exiting from proc with zero link; PROC VOID endprocess.

Words 13 to 28

- Flex-interrupt procedures (PROC VOID) in priority order.

Word 13	-	Keyboard interrupt.
Word 14	-	RS232 interrupt.
Word 15	-	GPIB interrupt.
Word 16	-	Tablet pointer interrupt.
Word 17	-	Winchester disc interrupt.
Word 18	-	Interval timer interrupt.
Word 19	-	Ethernet RX interrupt.
Word 20	-	Ethernet TX interrupt.
Word 21	-	Floppy disc interrupt.
Word 22	-	Speech interrupt.
Word 23	-	Breakin interrupt.

Words 24-28 unallocated.

9.4 Exceptions

Exceptions raised by the micro-code have characteristic of pairs of integers, the first being zero. Exceptions raised by program can have a characteristic consisting of any word pair. In particular,exceptions raised in D-state will produce (through successive calls of fail_proc in system_block) a chain of data giving diagnostic information about the exception and where it occurred. This chain is a reference to a six word object, the first four of which is the FAILQUAD given to the fail_proc procedure as parameter (cf workspace blocks):

1	2 bit gc	0	0010	0	3 bit spare	19 bit block address	1	:Failing codeblk
P	0000000000000			T	R	15 bit sf displacement	0	:Failing sf
12 bit rubbish				4 bit ls link, bpc		15 bit ms link, apc	0	:Posn of failure
1	2 bit gc	0	0001	0	3 bit spare	19 bit block address	1	:Failing wspace

and the remaining two is either a primitive exception or a reference to a similar object giving the failure in an inner procedure.

Micro-code exception pairs

Error(0,0)	- wrong type of value in U ; If the value in U is illegal, then the exception pair will come from the illegal.
Fail(0,1)	- index out of bounds.
Error(0,2)	- vector check fail (op code 58).
Fail(0,3)	- integer arithmetic overflow.
Error(0,4)	- wrong type of block.
Error(0,5)	- a or sz displacements wrong in some way,usually too big
Error(0,6)	- stack overflow in current work space.
Error(0,7)	- stack underflow in current work space.
Error(0,8)	- attempt to access outside a block.
Error(0,9)	- control value not allowed for in Case.
Error(0,10)	- attempt to use a pointer of the wrong sort in transput.
Error(0,11)	- attempt to use a privileged instruction without privilege.
Error(0,12)	- operand on tos is of wrong type.
Error(0,13)	- attempt to open keyed block with wrong key.
Error(0,14)	- attempt to access locked block.
Error(0,15)	- attempt to dereference nil.
Fail(0,16)	- real arithmetic overflow.
Error(0,17)	- illegal op code.
Error(0,18)	- attempt to dereference multiple chars or bools.
Error(0,19)	- attempt to read outside buffer.
Error(0,20)	- label out of scope in GOTO.
Error(0,21)	- attempt to call a non-procedure.

DOCUMENT CONTROL SHEET

Overall security classification of sheet ...Unclassified.......................................

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the box concerned must be marked to indicate the classification eg (R) (C) or (S))

1. DRIC Reference (if known)	2. Originator's Reference Report 85015	3. Agency Reference	4. Report Security Classification U/C
5. Originator's Code (if known)	6. Originator (Corporate Author) Name and Location Royal Signals and Radar Establishment		
5a. Sponsoring Agency's Code (if known)	6a. Sponsoring Agency (Contract Authority) Name and Location		
7. Title PerqFlex Firmware			
7a. Title in Foreign Language (in the case of translations)			
7b. Presented at (for conference papers) Title, place and date of conference			

8. Author 1 Surname, initials Currie IF	9(a) Author 2 Foster JM	9(b) Authors 3,4... Edwards PW	10. Date	pp. ref.
11. Contract Number	12. Period	13. Project	14. Other Reference	
15. Distribution statement Unlimited				

Descriptors (or keywords)

continue on separate piece of paper

Abstract

This report describes the instruction set and general firmware architecture of the Flex computor as implemented on the ICL Perq workstation

DATE
ILMED
5-8